

# Coding Practises

Or “Things that annoy me”

## **But my code ‘works’...**

- Ever asked for help?
- Ever sent your code to someone else?
- Ever put code online? (Do it, it’s great fun)
- Ever taken a break from a project and come back to it?
- Or just want to code more efficiently?

# Fundamentals

- Single Responsibility Principle
- Immutable vs mutable variables
- Decoupling of input/output
- Variable names that won't make you cry
- Minimising code duplication
- Code Linters

## **Single Responsibility Principle**

*Every class, function and method should handle **one** thing,  
and **one** thing only.*

# Single Responsibility Principle

Which is easier to read, understand, and update?

```
def get_data():
    supernovae = load_supernovae()
    supernovae['name'] = load_systematic('names.txt')
    supernovae['redshift'] = load_systematic('redshift.txt')
    supernovae['z'] = supernovae['redshift'].values
    supernovae['y'] = supernovae['name'].values
    supernovae['y'] = np.where(supernovae['y'] == 'supernovae', 1, 0)
    supernovae['y'] = np.where(supernovae['y'] == 'cosmology', 0, 1)
    supernovae['y'] = supernovae['y'].values
    return supernovae

def fit_cosmology(chains):
    filename_data = "supernovae.txt"
    filename_plot = "cosmology.png"

    supernovae_data = load_data(filename_data)
    chains = fit_cosmology(supernovae_data)
    plot_cosmology(chains, filename_plot)
```

**Horrifically Long Script**

```
def get_data():
    supernovae = load_supernovae()
    supernovae['name'] = load_systematic('names.txt')
    supernovae['redshift'] = load_systematic('redshift.txt')
    supernovae['z'] = supernovae['redshift'].values
    supernovae['y'] = supernovae['name'].values
    supernovae['y'] = np.where(supernovae['y'] == 'supernovae', 1, 0)
    supernovae['y'] = np.where(supernovae['y'] == 'cosmology', 0, 1)
    supernovae['y'] = supernovae['y'].values
    return supernovae

def fit_cosmology(chains):
    filename_data = "supernovae.txt"
    filename_plot = "cosmology.png"

    supernovae_data = load_data(filename_data)
    chains = fit_cosmology(supernovae_data)
    plot_cosmology(chains, filename_plot)
```

Makes it WAY easier to navigate code and contribute to it. Another big benefit is that it allows you to document code without writing documentation. By writing code that follows the SRP, often simply the function or class name is sufficient to explain what the code is doing. The right hand code's function can be arbitrarily long, but you still know what is going on!

Finally, its also a good way to get coding when you aren't sure how to start. List the tasks or things you need done. Great, now they're all functions. For each one, figure out the steps you need. And those will either be the lines of code, or more functions. At the end, with your code scaffolded out with empty functions, you can go back and fill in the blanks.

# Immutability

```
class Favourites(object):
    def __init__(self):
        self.set_default_favs()
    def get_favs(self):
        return self.favs
    def add_fav(self, fav):
        self.favs.append(fav)
    def set_default_favs(self, default=["pizza"]):
        self.favs = default

f = Favourites()
print(f.get_favs()) → ['pizza']

f.add_fav("chocolate")
print(f.get_favs()) → ['pizza', 'chocolate']

f.set_default_favs()
print(f.get_favs()) → ['pizza', 'chocolate']

f2 = Favourites()
print(f2.get_favs()) → ['pizza', 'chocolate']
```

A less contrived example, showing the effect in a given class instance, and the effect over separate instances

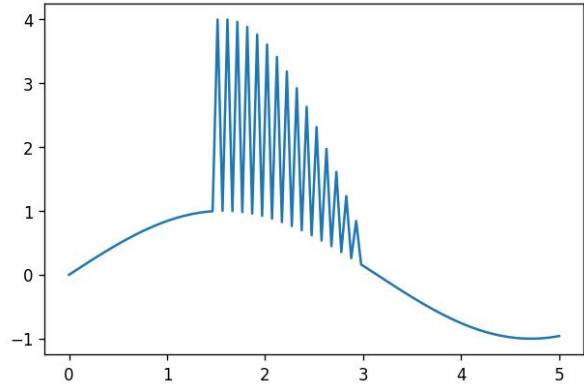
# Immutability

Be wary when passing mutable objects by reference.

```
import numpy as np
import matplotlib.pyplot as plt

def get_mult_sum(data):
    data *= 4
    return data.sum()

xs = np.linspace(0, 5, 100)
ys = np.sin(xs)
print(get_mult_sum(ys[30:60:2]))
plt.plot(xs, ys)
```



# Immutability

Can easily check by looking at the memory address:

```
# Immutable objects are fine, you get a new object
x = "Hello"
print(x, id(x))                                Hello 1950493571536
x += "!"
print(x, id(x))                                Hello! 1950493705304

import numpy as np
xs = np.linspace(0, 5, 100)
print(xs.__array_interface__)
print(xs[10::2].__array_interface__)
{'descr': [('', '<f8')], 'typestr': '<f8',
 'data': (1950474868240, False), 'version':
 3, 'shape': (100,), 'strides': None}
{'descr': [('', '<f8')], 'typestr': '<f8',
 'data': (1950474868320, False), 'version':
 3, 'shape': (45,), 'strides': (16,)}
```

Note that the last data address is 80 bytes down from the first in the array example. 80 bytes for 10 doubles. Means its super fast, but be careful. If the memory address changes on modification, its immutable. If it doesn't, mutable.

# Immutability

As primitives are immutable objects in python, don't have to worry about a lot of mutability issues such as you'd get in c.

Immutable objects:

- Int
- Float
- Bool
- String
- Tuple
- Range
- etc

Mutable objects:

- List
- Dict
- Set
- User defined classes
- bytearrays

So everything on the left is fine to go in a method signature, everything on the right should be avoided. And be careful when you expose mutable variables to other people / code, best to make a copy or trust that they dont change it!

# Decoupling Input

Note: Extremely simplified.

```
import numpy as np

def get_data():
    return np.load("data.npy")

def crunch_data():
    data = get_data()
    return data.sum(axis=0)

def save_results(data):
    np.savetxt("output.txt", data)

def analyse():
    crunched = crunch_data()
    save_results(crunched)
```

In a large projects these functions might be found hundreds of lines away or in different files.

So the idea is that input and output should be decoupled from algorithmic logic as much as possible. In this example, imagine trying to test `crunch_data` on a different dataset - you cant do it without editting some random `get_data` function

# Decoupling Input

Note: Extremely simplified.

```
import numpy as np

def get_data(file):
    return np.load(file)

def crunch_data(data):
    return data.sum(axis=0)

def save_results(data, file):
    np.savetxt(file, data)

def analyse(data_file, output_file):
    data = get_data(data_file)
    crunched = crunch_data(data)
    save_results(crunched)

analyse("test_data.npy", "test_output.txt")
```

Fully decoupled input and output.

Not always necessary - depends on what level  
your logic is.

So in the refactored example I've moved the input and output right to the top, but this isn't always necessary. If you have a function that uses data, but the rest of the application doesn't, just decouple that specific function. The goal is you should be able to quickly test whatever algorithm uses the data with a different dataset (such as a mock). Because this is such a simple example it goes right to the top.

The other advantage is you don't have to go hunting. Make the most likely to change code right at your fingertips

## Other notes

1. **Don't** redefine variables
2. **Don't** access globals from inside functions / classes
3. **Don't** write global code and also define a main statement
4. **Always** use **with** statements when you can... ie files
5. **Separate** plotting and analysis

```
# Bad
f = open("file.txt", "w")
f.write("Hello")
f.close()
```

```
# Good
with open("file.txt", "w") as f:
    f.write("Hello")
```

Other things to quickly point out

# **REASONS TO USE AN IDE**

# Variable Names and Reserved Words

Resist the temptation to just press a random key.

If you do, **refactor**. Don't waste your precious time doing it yourself!

Don't just avoid keywords, avoid shadowing builtins! If you don't want a variable, use an underscore.

```
import builtins  
dir(builtins)  
  
... Exception, False, None, True,  
Warning, abs, all, any, ascii, bin,  
bool, bytearray, bytes, callable, chr,  
compile, complex, dict, dir, enumerate,  
eval, exec, float, format, getattr,  
globals, hash, help, hex, id, input,  
int, isinstance, issubclass, iter, len,  
list, locals, map, max, min, next,  
object, oct, open, ord, pow, print,  
property, range, repr, reversed,  
round, set, slice, sorted, str, sum,  
super, tuple, type, vars, zip
```

Often bad variable names come through because I'm not sure if something is going to work, so write up really quick, low effort code to check. And if it does, it's easy to not go back and update things into legibility, and I end up with d, dd, d2, ddd all in one function.

This is why I recommend using an IDE like PyCharm. Got a bad variable name, just press Shift+F6 on the variable and rename it. The IDE will rename all usages of it for you, you don't need to go through your code and change them yourself, miss one, and thus break your code.

Mostly I just want to discourage people from overwriting the builtins. I have seen bugs introduced by this, even done it myself when I shadowed 'id' and then was doing some memory management stuff which invoked id.

# Code Duplication

Don't know how many times I've seen this sort of code:

```
import numpy as np
import matplotlib.pyplot as plt

def plot_all():
    file = "filename1.txt"
    data = np.loadtxt(file)
    plt.plot(data[:, 0], data[:, 1], label=file)
    plt.axvline(0)
    plt.xlim(0, 10)
    plt.legend()
    plt.savefig(file.replace(".txt", ".png"))

    file = "filename2.txt"
    data = np.loadtxt(file)
    plt.plot(data[:, 0], data[:, 1], label=file)
    plt.axvline(0)
    plt.xlim(0, 10)
    plt.legend()
    plt.savefig(file.replace(".txt", ".png"))

plot_all()
```

Minimise the work you have to do  
to refactor. Use an IDE.

Another recommendation for an IDE. Code duplication occurs because its easier to copy paste and check something works rather than refactoring yourself. But after it works, its hard to motivate yourself to go through and tidy up.

Modern IDEs have refactoring tools though that do it for you.

```
import numpy as np
import matplotlib.pyplot as plt

def plot_all():
    file = "filename1.txt"
    data = np.loadtxt(file)
    plt.plot(data[:, 0], data[:, 1], label=file)
    plt.axline(0)
    plt.xlim(0, 10)
    plt.legend()
    plt.savefig(file.replace(".txt", ".png"))

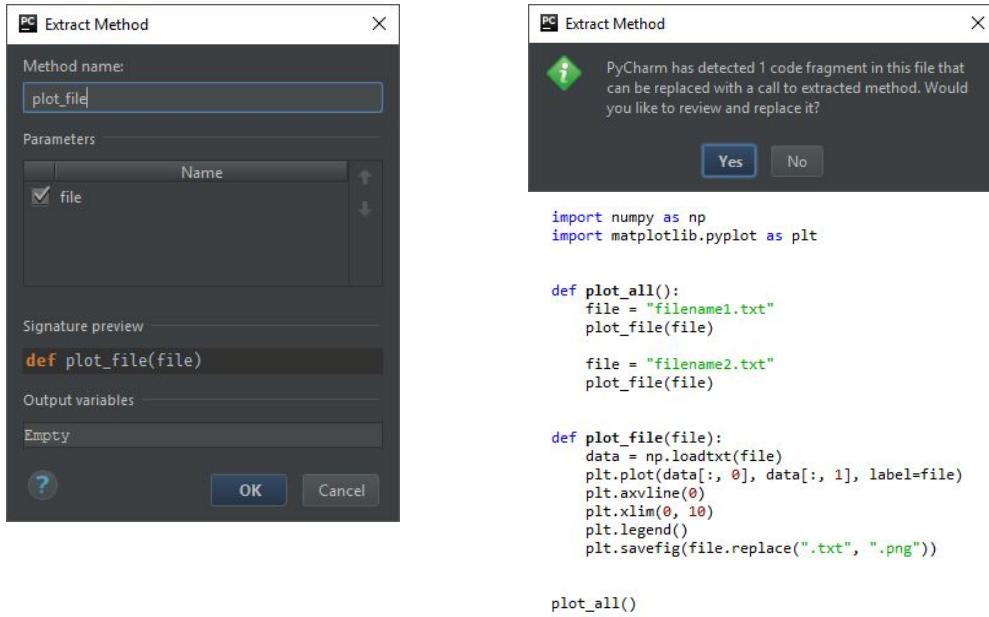
file = "filename2.txt"
data = np.loadtxt(file)
plt.plot(data[:, 0], data[:, 1], label=file)
plt.axline(0)
plt.xlim(0, 10)
plt.legend()
plt.savefig(file.replace(".txt", ".png"))

plot_all()
```

Cut Ctrl+X  
Copy Ctrl+C  
Copy as Plain Text  
Copy Reference Ctrl+Alt+Shift+C  
Paste Ctrl+V  
Paste from History... Ctrl+Shift+V  
Paste Simple Ctrl+Alt+Shift+V  
Column Selection Mode Alt+Shift+Insert  
Find Usages Alt+F7  
Refactor ►  
Rename... Shift+F6  
Change Signature... Ctrl+F6  
Folding ►  
Move... F6  
Copy...  
Safe Delete... Alt+Delete  
Extract ►  
Variable... Ctrl+Alt+V  
Constant... Ctrl+Alt+C  
Field... Ctrl+Alt+F  
Parameter... Ctrl+Alt+P  
Method... Ctrl+Alt+M  
Inline... Ctrl+Alt+N  
Pull Members Up...  
Push Members Down...  
Convert to Python Package  
Convert to Python Module  
Superclass...  
Subquery as CTE  
Local History ►  
Git ►

Another recommendation for an IDE. Code duplication occurs because its easier to copy paste and check something works rather than refactoring yourself. But after it works, its hard to motivate yourself to go through and tidy up.

Modern IDEs have refactoring tools though that do it for you.



Another recommendation for an IDE. Code duplication occurs because its easier to copy paste and check something works rather than refactoring yourself. But after it works, its hard to motivate yourself to go through and tidy up.

Modern IDEs have refactoring tools though that do it for you.

# Code Linters

Code linters will analyse the code you write to:

1. Improve readability
2. Pre-emptively detect bugs
3. Find unused code

Can run these manually, run them in a build pipeline (covered later) or use them integrated into an IDE

```
import numpy as np
def get_floor_max(a,b,c):
    if a>=b:
        maxv=a
    else:
        maxv=b
    sum=np.floor(maxv)
    return sum
if __name__=="__main__":
    d=get_floor_max(1,2,3)#Sums stuff
    print (d)
```

## Code Linters

PEP8 standard enabled by default in PyCharm

```
import numpy as np
def get_floor_max(a,b,c):
    if a>=b:
        maxv=a
    else:
        maxv=b
    sum=np.floor(maxv)
    return sum
if __name__=="__main__":
    d=get_floor_max(1,2,3)#Sums stuff
    print(d)
```



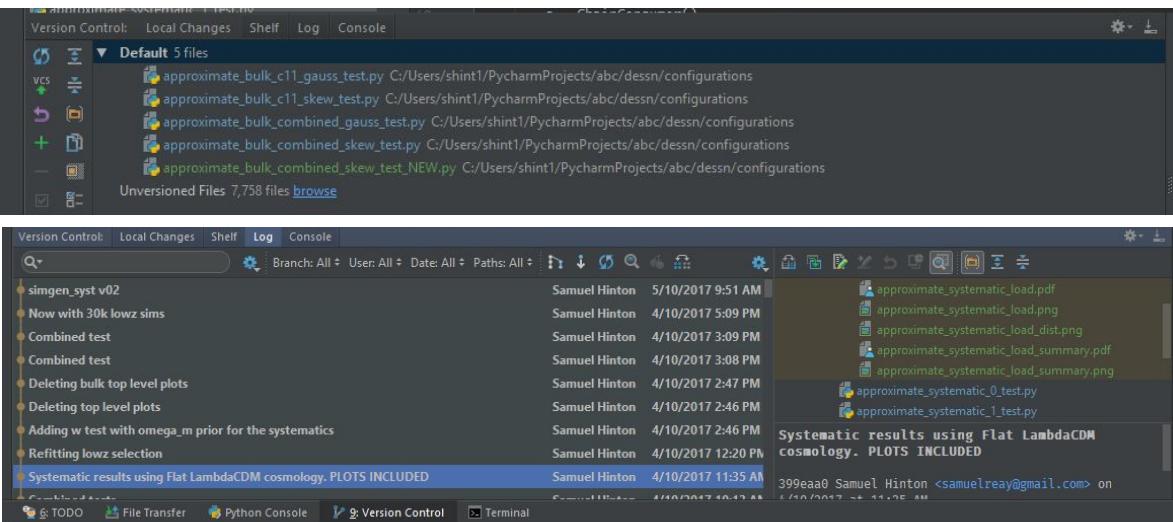
```
import numpy as np

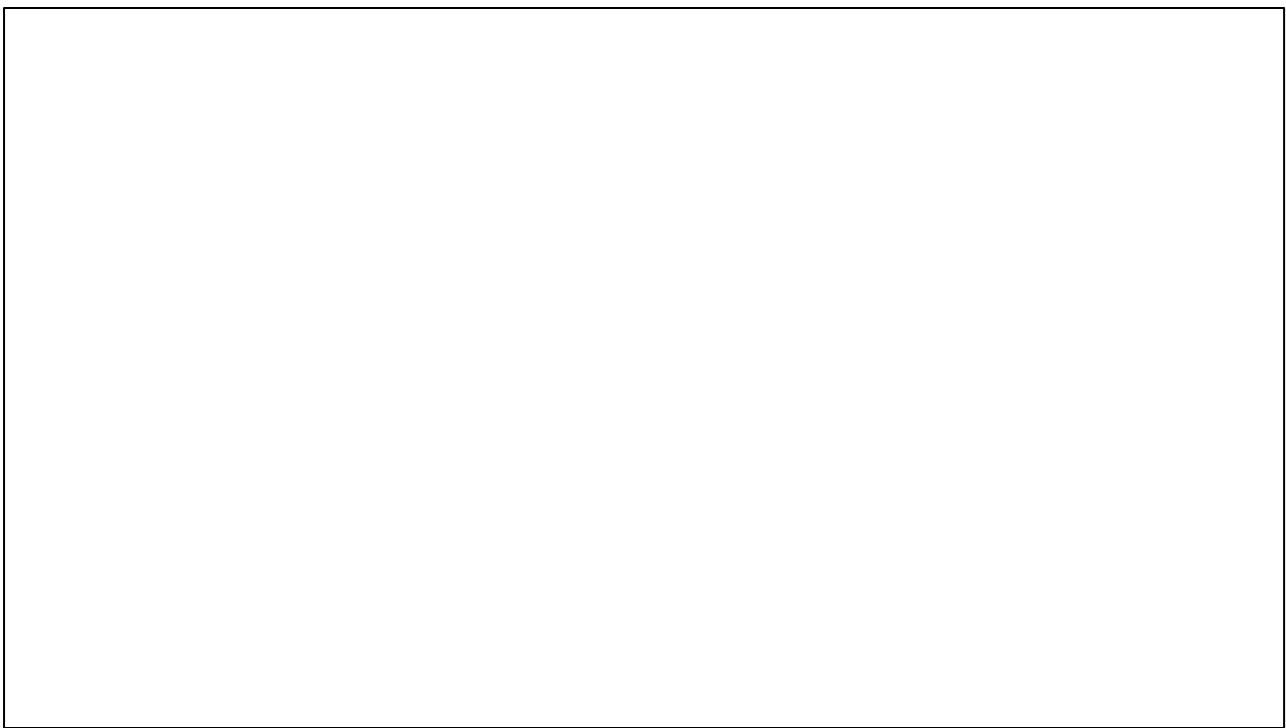
def get_floor_max(a, b, c):
    if a >= b:
        maxv = a
    else:
        maxv = b
    sum = np.floor(maxv)
    return sum

if __name__ == "__main__":
    d = get_floor_max(1, 2, 3) # Sums stuff
    print(d)
```

Can go to Code->Reformat file and have it format the code for you to take the hassle out of it. Note that some things like “c” and “sum” are still underlined. Can hover for why - c is not used, and sum shadows a builtin. IDEs really try to make your life simpler.

# Git Integration





```

import numpy as np
from numpy import *
f = open("/home/shint/6cols.txt", "r")
for line in f:
    headers = line.split()
    id, ra, dec, z, mag, mask = headers
    break

d = np.loadtxt("/home/shint/6cols.txt", skiprows=1)
mask = d[:, headers.index(mask)] == 1
zs = d[mask,headers.index(z)]
mags = d[mask, headers.index(mag)]
b = np.linspace(zs.min(), zs.max(), 10)
b[0] -= 0.000001
bc = 0.5*(b[1:]+b[:-1])
sums = array([0.0]*9)
cs = [0]*9
for i in range(len(zs)):
    z, mag = zs[i], mags[i]
    index = (z>b).sum()-1
    sums[index] += mag
    cs[index] += 1
print np.array(sums) / np.array(cs)

```

Q1: What does this code do?

Q2: How many ways can you improve this?

I guinea pigged this activity twice at Brisbane. I normally give five minutes to figure out what the code is doing, ask a few groups what they think, and then give between 5 and 10 minutes to come up with some updates. The next slide has the ones I find most annoying, but there are actually a few I considered too pedantic to list. Works well to go through each group asking what they would change in order or severity until you have gone through everyone and then jump to the next slide.

What the code does is calculate the mean magnitude for each redshift bin.

Yay	<code>import numpy as np from numpy import * f = open("/home/shint/6cols.txt", "r") for line in f:     headers = line.split()     id, ra, dec, z, mag, mask = headers     break</code>
Double import. Using * import	
Filename is OS / user dependent	
Assumes file structure and order	
Shadows id, unused vars	
Doesn't close file	
Duplicate filename	
Self shadowing, float/int comparison	
Inconsistent whitespace	
Redundant code	
Magic numbers (repeatedly)	
Dirty hacks	
Awful variable names	
Unpythonic declarations	
Mixing data types	
"Traditional" for loop	
Zip it	
Needlessly obtuse	
Amateur hour	
No return object. And... PYTHON2	<code>d = np.loadtxt("/home/shint/6cols.txt", skiprows=1) mask = d[:, headers.index(mask)] == 1 zs = d[mask, headers.index(z)] mags = d[mask, headers.index(mag)] b = np.linspace(zs.min(), zs.max(), 10) b[0] -= 0.00001 bc = 0.5*(b[1:]+b[:-1]) sums = array([0.0]*9) cs = [0]*9 for i in range(len(zs)):     z, mag = zs[i], mags[i]     index = (z&gt;b).sum()-1     sums[index] += mag     cs[index] += 1 print np.array(sums) / np.array(cs)</code>

Double import obviously bad

Star import is bad because you lose track of where things come from  
OS and user dependent filenames are a cardinal sin

The next section, assumes file structure to "Doesnt close file" is useless. Because I hardcode the order of the variables in the header assignment, theres literally no point doing this.

Refactored and cleaned code.

Of course, just use `scipy.stats.binned_statistic`

```
import numpy as np

def get_masked_data(filename):
    data = np.genfromtxt(filename, dtype=None, names=True)
    mask = data["mask"] == 1
    return data[mask]

def get_mean_per_bin(x, y, bins=9):
    bin_edges = np.linspace(x.min(), x.max(), bins + 1)
    # Map each entry to which x bin it belongs to
    mapping = np.digitize(x, bin_edges) - 1
    avg = np.array([np.mean(y[mapping == i]) for i in range(bins)])
    return bin_edges, avg

if __name__ == "__main__":
    data = get_masked_data("6cols.txt")
    _, avg = get_mean_per_bin(data["z"], data["mag"])
    print(avg)
```

So have broken things into functions to follow SRP. The `avg =` line requires stepping through it. Also good to draw attention to the destructive assignment in the second last line.

# **FUN PYTHON FEATURES**

# Logging

```
import logging
logging.basicConfig(level=logging.INFO)

def do_something(a,b):
    logging.info("<-- Look at the extra information!")
    return "%s %s" % (a, b)

print(do_something("No", "way"))
```

```
INFO:root:<-- Look at the extra information!
No way
```

As much fun as throwing a million print statements everywhere is, it can get confusing to keep track of them, especially if you're like me and often have `print("EUGH")`, `print("sdfghjksfdhjk")` and `print("1")`, `print("2")`, when you're desperately hunting for a bug. So use logging to keep track of all our output better!

The benefits of logging are: custom formats, differing log levels (critical, error, warning, info, debug), easily redirect them all into a different file and a ton more.

# Logging

```
import logging

fmt = '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
logging.basicConfig(level=logging.INFO, format=fmt)

class SomeClass(object):
    def __init__(self):
        self.logger = logging.getLogger(self.__class__.__name__)

    def do_something(self, a, b):
        self.logger.info("WHAT A USEFUL MESSAGE")
        if a == "No":
            self.logger.warning("WHY SO NEGATIVE?")
        return "%s %s" % (a, b)

print(SomeClass().do_something("No", "way"))
```

```
2017-11-16 12:45:47,521 - SomeClass - INFO - WHAT A USEFUL MESSAGE
2017-11-16 12:45:47,522 - SomeClass - WARNING - WHY SO NEGATIVE?
No way
```

As much fun as throwing a million print statements everywhere is, it can get confusing to keep track of them, especially if you're like me and often have `print("EUGH")`, `print("sdfghjksfdhjk")` and `print("1")`, `print("2")`, when you're desperately hunting for a bug. So use logging to keep track of all our output better!

The benefits of logging are: custom formats, differing log levels (critical, error, warning, info, debug), easily redirect them all into a different file and a ton more. In this example, I have a class logger which has the same name as the class itself. You can create new loggers whenever you want with different names, it doesn't have to be in the class

## Args and kwargs

```
def do_math(a, b, c):
    return a * (b ** c)

vals = [1, 2, 3]
print(do_math(vals[0], vals[1], vals[2]))
print(do_math(*vals))

def alt_max(*vals):
    # vals is a tuple
    return max(vals)

print(alt_max(3, 5, 7, 4, 3))
```

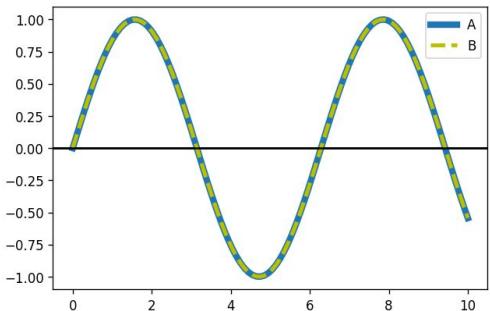
You can add args to functions and methods so they can take arbitrary number of arguments. However, the more common usecase is to use the splat operator to unwrap a list of options when the function you are calling wants them separate.

## Args and kwargs

```
def plot_with_options(x, y, label="Data", **kwargs):
    plt.plot(x, y, label=label, **kwargs)
    plt.axhline(0, c='k')
    plt.legend()

x = np.linspace(0, 10, 100)
y = np.sin(x)
opts = {"c": "y", "ls": "--"}

plot_with_options(x, y, lw=5.0, label="A")
plot_with_options(x, y, label="B", lw=3.0, **opts)
```



Like args, there are kwargs. Example is chosen to show both how they can be used in a function signature, and how you can unpack dictionaries to pass them in as kwargs. The example use is something I see a lot of, where plotting calls or similar have duplicate entries (like all plots are made with linewidth thicker for example). Instead of having to change each call if you decide to update, you can put the common arguments into a dictionary and then unpack them. Included the linewidth in the “B” plot call to illustrate that you can combine inline keywords with unpacked keywords, you dont need to have one or the other.

## Args and kwargs - tools of denial

```
# Almost everyone does this
x = np.random.normal(0, 1, 1000)
hist, bins = np.histogram(x, 100)

# When the actual syntax uses kwargs
x = np.random.normal(loc=0, scale=1, size=1000)
hist, bins = np.histogram(x, bins=100)

# There are dangers, so can shut them down like this
def histogram(data, *args, bins=10):
    assert len(args) == 0, "WAT U DOIN?"
    return np.histogram(data, bins=bins)

hist, bins = histogram(x, bins=100)
hist, bins = histogram(x, 100) # ASSERTION ERROR
```

A feature of python is that if you finish specifying args and there are kwargs left, you can just keep specifying them without the keywords. HOWEVER, if any future update reorders the keywords, or adds a new one not at the end, or if there are any more arguments added, all previous code that doesn't specify the keywords will break.

If this might happen to code you are writing, you can deny this feature by routing all the non-keyword extra arguments into a tuple of \*args - the only way to set the kwargs is then to use the keywords.

## Unpacking (more splat)

Whilst we're splatting, you can use them  
to unpack better in Python3.

```
h, e, l, _, o = "hello"  
h, *ell, o = "hello"  
*hell, o = "hello"
```

Underscore is a placeholder, not a variable name. Use it to discard variables you dont want. Variable names indicate whats in them.

# Generators

```
xs = range(1000000000) # ONE BILLION NUMBERS PLS
```

With Python3, you can run this. Python2, you will get a MemoryError

Traditional

```
def fibonacci(n):
    res = [0, 1]
    for i in range(2, n):
        res.append(res[-1] + res[-2])
    return res

for number in fibonacci(10):
    print(number)
```

Generator

```
def fibonacci2(n):
    a, b = 0, 1
    for i in range(n):
        yield a
        a, b = b, a + b

for number in fibonacci2(10):
    print(number)
```

Generator v2

```
def fibonacci3():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b
```

So python2 allocates a huge array, runs of memory, and crashes. Generators dont preallocate everything, they only give you what you want when you ask for it (when you iterate through it)

Yes the traditional assumes  $n > 2$ .

Two different ways of generating fibonacci numbers. The first is using the traditional approach like you would in languages like c. If you call this with a large n, like 10000, you will get a big hangup initially and then itll print everything out at once. With the generator method you start printing immediately, because you dont have to precompute everything. You only calculate the nth number when you request it, so its more efficient. The third example is showing that because of the stateful nature of the iterator you dont need to set an upper limit, you can just keep calling it however many times you like.

## f-strings

Welcome to python 3.6

```
user = "Sam"  
num = 20  
cool_string = f"User {user} has {num} bugs to fix. Damn"  
print(cool_string)
```

User Sam has 20 bugs to fix. Damn

```
name, age = "samuel hinton", 25  
print(f"{name.title()} is {12*age} months old.")
```

Samuel Hinton is 300 months old.

```
good = ["pizza", "chocolate"]  
name = "Bob"  
print(f"{name} likes {good[0]} the most")
```

Bob likes pizza the most

f-strings are amazing, but youll need the latest python to get them. You can do a lot in the braces, because its evaluated directly (and cast to string). No more temp variables you just create for string formatting, and no more verbose formatting!

## lru\_cache

```
from functools import lru_cache

def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

@lru_cache(maxsize=None)
def fib2(n):
    if n < 2:
        return n
    return fib2(n-1) + fib2(n-2)

%time fib(35)
%time fib2(35)
print(fib2.cache_info())
```

Wall time: 3.94 s  
Wall time: 0 ns  
CacheInfo(hits=33, misses=36,  
 maxsize=None, currsize=36)

What a beauty.

So lru\_cache caches the last n (maxsize) number of calls to the function and the output. Set maxsize to a power of 2, or leave it to None to let it grow forever! If you have a stateless function you call a lot throwing the lru\_cache annotation on it will speed things up a ton!

# Decorators

```
import time

def timed(fn):
    def timer(*args, **kwargs):
        t_start = time.time()
        result = fn(*args, **kwargs)
        tdiff = time.time() - t_start
        print("%s took %0.2f sec" %
              (fn.__name__, tdiff))
        return result
    return timer

@timed
def stupid_sum():
    x = 0
    for i in range(10000000):
        x += i
    return x

val = stupid_sum()

def counted(fn):
    def inner(*args, **kwargs):
        inner.count += 1
        print("%s called %d times" %
              (fn.__name__, inner.count))
        return fn(*args, **kwargs)
    inner.count = 0
    return inner

@counted
def blah():
    pass

#blah = counted(blah)

for i in range(5):
    blah()
```

Right, so decorators are something I never see used but can actually be useful. Here are two simple examples, one to time a function, and another that counts the number of times a function is invoked.

The commented out line on the right is a way to think of what is happening. The decorator overwrites the original function pointer to a point of the inner function returned. That inner function generally invokes the original function (although you can make a decorator which calls a separate function instead or does nothing if you want). On the left is a stock standard function, on the right we use the fact a function is an object and attach a variable to the function called count.

These decorators are examples and not very useful, useful ones are like lru\_cache

# **Summary**

Write good code.

**The end.**

# **Forking a Github Repo**

# Setting up a project

A small github repo used as an example for the coding workshop

Add topics

Branch: master ▾ New pull request

Create new file Upload files Find file Clone or download ▾

Samreay Updating travis	Latest commit e63e0f2 6 minutes ago
some_module	Basic layout
tests	Basic layout
.gitignore	Basic layout
.travis.yml	Updating travis
README.md	Initial commit
requirements.txt	Basic layout
setup.py	Adding setup.py

Made a Github project to show a build up. There is some module, and some tests, with the normal files - gitignore, setup.py, requirements.txt. Lets run through them real quick.

# Setting up a project

Branch: master ▾ [WorkshopExample](#) / [.gitignore](#) Find file Copy path

 Samreay Basic layout 629e5ae 24 minutes ago

1 contributor

103 lines (81 sloc) | 1.14 KB Raw Blame History

```
1 # Byte-compiled / optimized / DLL files
2 __pycache__/
3 *.py[cod]
4 *$py.class
5
6 # C extensions
7 *.so
8
9 # Distribution / packaging
10 .Python
11 env/
12 build/
13 develop-eggs/
14 dist/
15 downloads/
16 *.egg/
```

Gitignore. Github will generate this for you if you pick a language when you create the repo.

# Setting up a project

Branch: master ▾ [WorkshopExample / requirements.txt](#)

 Samreay [Changing requirements](#)

1 contributor

8 lines (7 sloc) | 65 Bytes

```
1 numpy
2 pytest
3 pytest-cov
4 codecov
5 sphinx
6 numpydoc
7 sphinx_rtd_theme
```

Requirements file. Should list every dependency you need to install to get your project running, tested, documented. Everything.

Branch: master ▾ [WorkshopExample / setup.py](#)

 **Samreay** Changing requirements

1 contributor

13 lines (11 sloc) | 408 Bytes

```
1 from setuptools import setup, find_packages
2
3 setup(name='WorkshopExample',
4       version='0.0.1',
5       description='Random example project for coding workshop',
6       url='http://github.com/Samreay/WorkshopExample',
7       author='Samuel Hinton',
8       author_email='samuelreay@gmail.com',
9       license='MIT',
10      install_requires=['numpy'],
11      packages=find_packages(exclude=('tests', 'doc'))
12 )
```

Okay, so most of this is self explanatory. Two things which are worth pointing out - normally you can just write the packages (in this case, it would be `some_module`), but to make it more general Im using the `find_packages` function, and excluding the tests and doc directory (doc doesnt exist right now, but soon).

The main thing to point out here is the difference between `requirements.txt` and `install_requirements`. `Install_requirements` should document the most general, loosest possible requirements for installing and running the package. This is what PyPI uses to install dependencies when you `pip install`. `Requirements.txt` on the other hand is meant to declare “Using these specific packages/versions, you can define a working development/test environment”. So one is general user focused, and one is dev focused.

# ***Task: Forking a Github Project***

<https://github.com/Samreay/WorkshopExample>

On Github, everyone fork the project, clone it using PyCharm and commit a small change.

Follow the README steps 1 to 3 on the repo .

Ensure you find/replace my username with yours at the very least.

And now, lets get everyone to open this repo, fork it, clone it and get commits working. What I found useful to do is grab the laptop of someone already familiar with git/python (that would be able to do this on their own) and do it live with their github account.

The important step for them to do initial is to find and replace (right click WorkshopExample in Pycharm, replace in path, samreay -> their\_username, done) my username with theirs, so that things point to their repo and not mine.

And then onto code testing

# **Code Testing**

Fail hard. Fail fast.

## ***Code without guilt or worry***

This is the main point - modify, change, add, remove, do whatever you want with your code and dont worry about breaking current features or current workflow when it might impact your results, youll know if anything changes right away and diagnose issues right where the change has happened instead of spending days tracking down an issue because a number spat out at the end of your analysis is now wrong.

## Fundamentals

- Unit tests vs integration tests
- How to test
- Testing pipelines and utilities

# Categories

## Unit Tests

- One test on the *smallest* scale
- Each test tests *one* thing
- Cheap and fast to run

## Integration Tests

- Tests how pieces interconnect
- Tests from *start to end*
- Slow and expensive to run

So just a brief overview of the difference between unit tests and integration tests so that I can quickly go straight to unit tests and the context is clear

## Unit Tests

```
def count_occurrences(search, document):
    """ Count the occurrences of one string in another, case insensitive

    Finds the number of occurrences, including overlapping cases,
    for how many times one string is present inside another. Uses
    a simple brute force algorithm, so try not to pass in massive strings

    Parameters
    -----
    search : str
        The string to look for
    document : str
        The string to look for the `search` term inside

    Returns
    -----
    int
        The number of occurrences

    Examples
    -----
    >>> count_occurrences("bob", "Hey Bob, Bobby, Bobo, Bobobo")
    5

    """
    count = 0
    document = document.lower()
    n = len(search)
    for i in range(len(document)):
        if document[i:i+n] == search:
            count += 1
    return count
```

So let's be clear, such a simple method doesn't need so much docstring, but for posterity and repeating how to do it, here it is. But as to "What does this method do" we now read a sentence instead of a section of code.

In terms of what should be tested, the obvious one here is to test the given example. There are at least two bugs in the above code, one of which I'll point out and fix in later slides (empty string search matches everything, the other is also fairly obvious - search should be cast to lowercase but isn't). The point is eventually we will get 100% test coverage on our code and yet there is still a bug, because 100% coverage != 100% tested

## Unit Tests

How would you test the following function?

```
def count_occurrence(search, document):
    count = 0
    document = document.lowercase()
    n = len(search)
    for i in range(len(document)):
        if document[i:i+n] == search:
            count += 1
    return count
```

So thought it would be good to jump straight in. The function above just counts the number of occurrences of one string inside another by brute force. But dont spend time on this slide, jump to the next because documentation, reinforce the point of why doco is useful.

# Unit Tests

General Principle: **0, 1 and many**

Catch the edge cases!

```
def test_count_occurrence_simple():
    assert count_occurrences("bob", "Hey Bob, Bobby, Bobo, Bobobo") == 5

def test_count_occurrence_letter():
    assert count_occurrences("a", "a") == 1

def test_count_occurrence_empty_search():
    assert count_occurrences("", "hello") == 0

def test_count_occurrence_empty_document():
    assert count_occurrences("a", "") == 0

def test_count_occurrence_empty_both():
    assert count_occurrences("", "") == 0
```

So lets say we come up with these five test cases to start with, but havent run them yet. Now the point I want to get across here is that \*we dont need to method to exist to write the test cases\*. In some cases in code development, you write the tests **\*before\*** you write the algorithm itself, because the tests should define how the function acts. The above list of five isnt an exhaustive list of tests, Ill add something a bit more complicated later on, but these are basic tests that we can probably all agree on. Test case sensitivity, multiple matches, single matches, empty string permutations. Have no tested type checking.

The other important thing to note is that unit tests need to have more coverage on API or external methods, its not necessary to unit test every function you ever write, and in some cases for internal functions the behaviour may be explicitly undefined if the inputs are wrong (type/value, etc). Because those functions are called by your code, and not an unknown user, you dont need to be as defense.

## Running Unit Tests

```
test_count_occurrence_simple()  
test_count_occurrence_letter()  
test_count_occurrence_empty_search()  
test_count_occurrence_empty_document()  
test_count_occurrence_empty_both()
```



Never, never do this.

No metrics. Will abort on first failure. **WILL NOT IMPORT.**



## Running Unit Tests

```
if __name__ == "__main__":
    test_count_occurrence_simple()
    test_count_occurrence_letter()
    test_count_occurrence_empty_search()
    test_count_occurrence_empty_document()
    test_count_occurrence_empty_both()
```



Never, never do this.

No metrics. Will abort on first failure.

# Running Unit Tests With PyTest

```
pytest code.py      if not on path:  python -m pytest -v code.py
```

```
===== test session starts =====
platform win32 -- Python 3.5.2, pytest-2.9.2, py-1.4.31, pluggy-0.3.1 -- C:\Anaconda3\python.exe
cachedir: .cache
rootdir: C:\Users\shint1\Google Drive\Docs\2017 CAASTRO Workshop Proposal\3 Code Testing, inifile:
plugins: cov-2.4.0
collected 5 items

code.py::test_count_occurrence_simple PASSED
code.py::test_count_occurrence_letter PASSED
code.py::test_count_occurrence_empty_search FAILED
code.py::test_count_occurrence_empty_document PASSED
code.py::test_count_occurrence_empty_both PASSED

===== FAILURES =====
____ test_count_occurrence_empty_search ____

def test_count_occurrence_empty_search():
>     assert count_occurrences("", "hello") == 0
E     assert 5 == 0
E     +  where 5 = count_occurrences('', 'hello')

code.py:54: AssertionError
===== 1 failed, 4 passed in 0.04 seconds =====
```

Yay for pytest

So you don't need to write anything other than the functions in your file. Pytest will look through the file, and run any function that starts with "test\_". It will do the same for classes like "TestClass" and execute "test\_\*" methods. The -v is for verbose so you can see the passed ones as well. See that it runs all tests, and instead of just saying "AssertionError" the previous two slides, it actually breaks down why the assertion failed! How useful!

Actually they don't even need to be in the same file. We can remove the "code.py" part of our call, and make a file called test\_code.py, and because it matches the "test\_" discovery pattern pytest will pick it up automatically. (Next slide has those patterns on it)

# PyTest Discovery Rules

<https://docs.pytest.org/en/latest/goodpractices.html#test-discovery>

`test_*.py`

`*_test.py`

(Recursive directory search)

```
def test_*
class Test*:
    def test_*
```

# Code Coverage

So let's break things up:

example

\_\_init\_\_.py

code.py

test\_code.py

```
def count_occurrences(search, document):
    if not search:
        return 0
    count = 0
    document = document.lower()
    n = len(search)
    for i in range(len(document)):
        if document[i:i+n] == search:
            count += 1
    return count

from .code import count_occurrences

def test_count_occurrence_simple():
    assert count_occurrences("bob", "Hey Bob, Bobby, Bobo, Bobobo") == 5

def test_count_occurrence_letter():
    assert count_occurrences("a", "a") == 1

def test_count_occurrence_empty_document():
    assert count_occurrences("a", "") == 0
```

Using the previous example, lets modify two things. First, to deal with that test failure, lets check for search being empty, and return 0 if that is the case. Secondly, Im going to remove the two tests which actually used an empty search, so I can demonstrate code coverage better. Ive also put the tests in a separate file, and put everything in a folder with an \_\_init\_\_.py, so that we have an example module. Might be worthwhile here to explain what \_\_init\_\_.py does - it tells python to treat that directory as a module (ie treat it as a source directory and let imports work, etc)

# Code Coverage

```
python -m pytest --cov-report html --verbose --cov example
```

Needs `pytest-cov` to be installed, but we can now say “Give us an HTML coverage report of the example module.” And suddenly a `htmlcov` directory exists and we open the `index.html`:

Coverage for `example\code.py` : 90%

10 statements 9 run 1 missing 0 excluded

```
1 def count_occurrences(search, document):
2     if not search:
3         return 0
4     count = 0
5     document = document.lower()
6     n = len(search)
7     for i in range(len(document)):
8         if document[i:i+n] == search:
9             count += 1
10    return count
11
```

So we can ask for an html coverage report with the command above, and it'll show you all the files and how much of the code is covered. We can see that by removing those two test cases our tests never enter the `if` statement, and so the coverage report is telling us to add them back in! In the main `index.html` it'll also show the coverage for the test file (which should be 100% because `pytest` ran all the tests). However, this isn't a useful file to show, and we can specify which files to compute coverage on using a `.coveragerc` file and excluding anything with `test` in the name, or exclude a test directory, etc.

## Overdoing it

```
def count_occurrences(search, document):
    assert isinstance(search, str), \
        "search should be a string"
    assert isinstance(document, str), \
        "document should be a string"
    if not search:
        return 0
    count = 0
    document = document.lower()
    n = len(search)
    for i in range(len(document)):
        if document[i:i+n] == search:
            count += 1
    return count
```

```
from .code import count_occurrences
import pytest

def test_count_occurrence_simple():
    assert count_occurrences("bob", "Bob,Bobby,Bo")

def test_count_occurrence_letter():
    assert count_occurrences("a", "a") == 1

def test_count_occurrence_empty_document():
    assert count_occurrences("a", "") == 0

def test_count_occurrence_empty_search():
    assert count_occurrences("", "hello") == 0

def test_count_occurrence_empty_both():
    assert count_occurrences("", "") == 0

def test_count_occurrence_not_string_search():
    with pytest.raises(AssertionError):
        count_occurrences(0, "hello")

def test_count_occurrence_not_string_document():
    with pytest.raises(AssertionError):
        count_occurrences("a", {})
```

So here are a set of tests which will get 100% code coverage. Its probably also over the top, imagine trying to do this for \*every\* function you write - its just not going to happen. The style of coding shown on the left, type checking, boundary checking, etc, is known as defensive coding, and it only viable or needed in specific circumstances.

Note here we have 100% test coverage, but there is still a bug - search is never cast to lwoer

## Defensive Coding

Trust nothing. Plan for everything.

Many sorts of defensive coding, but main one in astro context is “Validating input”. Assume you will be given garbage, and either check for garbage or ensure your algorithm can handle it.

**ONLY** worth the effort for external APIs.

*Private functions are rarely worth the effort!*

So defensive coding is needed in high stake applications where you need to make sure nothing the user does crashes your program (like trading on the stockmarket). But this isn't too relevant to our use cases, most of what we want to do testing for is for our own benefit. Ie if we write a numerical integration function, to test it on functions with analytic solutions so that we can gain confidence in our own code.

# Integration Tests

Relevant for projects with multiple connecting pieces, where you test how the pieces interact.

Each individual component should already be unit tested.

```
class DataLoader(object):
    def load_data(self, filename):
        pass # Return data

class Parser(object):
    def parse_data(self, data):
        pass # Return some parsed data

def test_data_integration():
    filename = 'test_data.txt'
    data_loader = DataLoader()
    data = data_loader.load_data(filename)
    parser = Parser()
    parsed_data = parser.parse_data(data)
    assert parsed_data is not None
    # Assert parsed_data is a valid product
```

Integration testing isn't too common or needed in astro. But if you're making a data pipeline with multiple components, this would be what you want to have! Or anything which properly separates out things like data loaders, serialisers, and an analysis. Integration testing is, as the name suggests, about testing how the different pieces you have constructed fit and integrate together.

## **System Tests**

Consider a project like a data pipeline. A system test would validate the entire pipeline, feeding in test data and validating the output products.

The layer above this integration testing is system testing, where you essentially test everything, start to finish. Ie a data pipeline would have system testing where fake 'raw telescope data' is fed in one end, and processed galaxy properties come out the other. Useful normally only in larger projects, would not be done for scripts.

## **Summary**

Many ways to test:

- Unit tests
- Integration tests
- System tests

Easy to setup pytest to run tests for you. So do it.

Will do an interactive testing activity at the end to incorporate build pipelines

So there are a few slides, and then when you get to Extending Travis title slide, lets do it interactively. Check speaker notes in that title slide.

# Continuous Integration Options

## GitLab

Integrated internal solution:

[https://docs.gitlab.com/ee/ci/quick\\_start/](https://docs.gitlab.com/ee/ci/quick_start/)

## GitHub

Travis: <https://travis-ci.org/>

CircleCI: <https://circleci.com/>

Jenkins: <https://jenkins.io/>

So the idea that you make a change in your repo and its immediately picked up and tested is known as continuous integration. There are a few options here, but tl;dr we're going with travis.

Gitlab has their own integrated solution, but have decided to focus on GitHub at the moment. There are multiple options for GitHub: Travis, CircleCI, Jenkins (the big three). Travis is in green because I will focus on that. CircleCI is very similar (similar ways to configure, similar features, both cloud based). Jenkins (Travis and Jenkins used to be the same project a long time ago before it split) requires a dedicated server (it's a much more heavyweight solution)

# Setting up Travis

1. Click a box



2. Add a .travis.yml file

```
language: python

python:
  - "2.7"
  - "3.6"

install:
  - pip install .
  - pip install -r requirements.txt

script:
  - pytest
```

<https://github.com/Samreay/WorkshopExample/tree/e63e0f22de017db2b350329119093c32706d3246>

Getting travis to track a project is fairly easy. Sign into travis with your github account, check the box that activates travis on your project, and add a .travis.yml file to the top project dir. The url down the bottom is a link to a git example repo with minimal content with a working travis integration.

The .travis.yml file which goes in the top directory of the project shown here is very simple. We declare what language we are working in, we say what version(s) of python we want to build with (so this config will spawn two parallel builds, one for 2.7 one for 3.6). Then the install commands get run. Requirements.txt should contain everything your project needs to run. Note that because of the way I have linked them together in the setup.py, the first pip install will install everything in requirements, but good to put both in case you decide to break the link.

I dont put pytest, sphinx, etc in the requirements.txt because you dont need them to actually run the code, so keep them separate - either its own file or explicitly in .travis.yml

The script that gets run simply executes pytest

Note that the travis.yml in the repo has more stuff in it, so that noone has to update it as the day goes on. But this would be the minimal travis config you need for pytest.

# Setting up Travis

Samreay / WorkshopExample  build unknown

Current Branches Build History Pull Requests

More options 

✓ master Updating travis	→ #3 passed	 Restart build
→ Commit e63e0f2 ↵	⌚ Ran for 49 sec	
↳ Compare df39c22..e63e0f2 ↵	⌚ Total time 1 min 29 sec	
↳ Branch master ↵		
(S) samreay authored and committed	🕒 3 minutes ago	

Build Jobs

✓ # 3.1	 Python: 2.7	📦 no environment variables set	⌚ 49 sec	
✓ # 3.2	 Python: 3.6	📦 no environment variables set	⌚ 40 sec	

And this is the output from travis. Can click on either 3.1 or 3.2 builds below to see the detailed logs.

# Example Failure

WHAT A FOOLISH MISTAKE

master

Samreay committed 28 seconds ago

1 parent 9725fe9 commit e9312b518be6fbea68e5c409eae5eccb9390d41a

Showing 1 changed file with 1 addition and 0 deletions.

Unified Split

View ▾

```
1 some_module/file1.py
...
... @@ -1,4 +1,5 @@
1   1
2   2
3   3     def add_stuff(a, b):
4 +     print a
4   5     return a + b
```

So what happens when we screw up? So here I put myself in the shoes of someone developing in python2 and not realising the project has to also be python3 compatible, so I put in a bad print statement. Well, as soon as I push the changes to github, travis kicks off

# Example Failure

Samreay / WorkshopExample  build failing

Current Branches Build History Pull Requests More options 

 master WHAT A FOOLISH MISTAKE	 Commit e9312b5 ↗	 Compare 9725fe9..e9312b5 ↗	 Branch master ↗	 samreay authored and committed	 #6 failed	 Ran for 47 sec	 Total time 1 min 20 sec	 about a minute ago	
---	--	--	---	--	---	--	---	--	---

Build Jobs

 # 6.1	 Python: 2.7	 no environment variables set	 34 sec	
 # 6.2	 Python: 3.6	 no environment variables set	 46 sec	

And sure enough, the 3.6 python build fails hard! Can click on the failed run to get the log, and it'll be an error about lack of brackets.

# Extending Travis 1

## *Adding Code Coverage*

So lets do the previous section interactively now. Get everyone to log into travis with their github credentials, enable travis on the repo (they may have to resync accounts and refresh to see it). Travis might not kick off a build right away, so just commit some menial change to get the build started. Once a build has started for everyone, good to move on.

## Extending Travis - Adding Code Coverage

Using [CodeCov](#)

<https://codecov.io/>



Other options exist (coveralls,  
Code Climate, more).



So quite a few different platforms can be used, but the one I am going to run with is CodeCov. I believe it is the best solution out there at this point in time (better support for python, better integration, better layout, easier to set up).

# Extending Travis - Adding Code Coverage

1 <https://codecov.io/>

A screenshot of the codecov.io homepage. At the top, there's a navigation bar with links for 'gh', 'Samreay', 'Docs', 'Support', and a user profile picture. Below the navigation, there's a main area with a 'Repositories' tab selected. A message box says 'Codecov currently has public-only scope only. Please add private repository scope to view and add private repositories.' with a 'Hide' button. Another message box below it says 'Create a subscription to add Private Repositories. Click here to upgrade your plan.' with a 'Hide' button. At the bottom right of this area is a green button labeled 'Add new repository'. A red number '2' is overlaid on this button.

A screenshot of a project page on codecov.io. The top navigation shows the repository 'Wigglez' and 'WorkshopExample'. A red number '3' is overlaid on the 'WorkshopExample' link. Below the navigation, there's a large green button with the text 'Hey! Let's get your project covered.' A sub-section titled 'STEP 1 - COPY TOKEN' shows a token field containing '88c72757-a76e-470f-aaf7-5e6cf2be5d01' with a 'Copy' button next to it. A note below says 'Use this token when uploading reports. You'll see how in the example repository below.'

Setting it up is fairly easy. Log into codecov using github/bitbucket/gitlab. Hit “Add new repository” (2), pick the repo (3), and then if you’re not using Travis youll need to use the token as they describe at the site (which I didnt include in the image). For Travis you dont need to set this up, its automatic.

To repeat: Step 4 does not need you do anything. As soon as you see the green dude with the umbrella at the top of your screen, you are done. Travis doesnt need copying the token.

## Extending Travis - Adding Code Coverage (2)

5

Branch: master ▾ WorkshopExample / .coveragerc

Find file Copy path

15 lines (14 sloc) | 209 Bytes

Raw Blame History

```
1 [report]
2 omit =
3   doc/*
4   build/*
5   dist/*
6   examples/*
7   tests/*
8   *__init__
9   setup.py
10
11 exclude_lines =
12   pragma: no cover
13   raise NotImplementedError
14   raise ImportError
15   def __len__
```

We should also set up a `.coveragerc` file, which you can use to set up what will be analyses. Here I've gone and done more than needed for the current project (ie there is no build or dist directory yet), but this is so that you can essentially use this file even in other projects and not worry about changing things.

## Extending Travis - Adding Code Coverage (2)

### 6 Update .travis.yml

```
language: python

python:
  - "2.7"
  - "3.6"

install:
  - pip install .
  - pip install -r requirements.txt


script:
  - pytest --cov=./ -v

after_success:
  - codecov
```

So we tell travis to run code coverage on everything (normally you'd put the package name here, not ./, but trying to keep this as general as possible). And if the script is successful, calling codecov will upload the results to CodeCov.

## 7 Commit and confirm

Files	4	3	0	1	Coverage
some_module/file1.py	4	3	0	1	75.00%
Project Totals (1 files)	4	3	0	1	75.00%

## 8 [Optional] Funky badge

A screenshot of a GitHub commit page for a file named README.md. The commit message contains the following text:

```
... ... @@ -1,2 +1,5 @@
1 1 # WorkshopExample
2 +
3 +![Coverage Status](https://codecov.io/gh/Samreay/WorkshopExample/branch/master/graph/badge.svg)
4 +
2 5 A small github repo used as an example for the coding workshop
```

At the bottom right of the commit message, there is a "codecov" badge with the text "75%".

Best to now check the travis logs and codeCov to make sure it all works. Yay. And of course everything should have a badge

# Done

So lets do the previous section interactively now (again). Get everyone to log into codecov with their github credentials, enable codecov on the repo. Tell travis to rebuild, and soon they should have a report in codecov.io

# **Code Documentation**

Just do it

## **But my code ‘works’...**

- Ever asked for help?
- Ever sent your code to someone else?
- Ever put code online? (Do it, it’s great fun)
- Ever taken a break from a project and come back to it?
- ~~Or just want to code more efficiently?~~

## Fundamentals

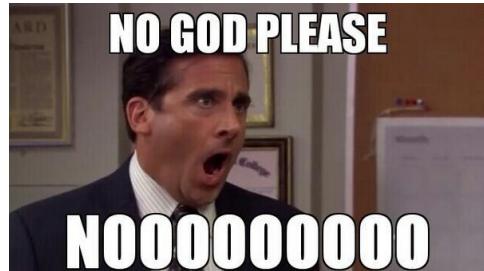
- What Not To Do
- Self Documenting Code
- Comments vs Docstring
- How to Docstring

Signposting.

## What Not To Do

If you've ever seen an *Introduction to Coding* assignment with marks for code comments you know where I'm going.

```
# initialise variables
acc = 0
# Loop over all items
for item in alist:
    # Add them up
    acc += item
# end Loop
```



Comments should never turn basic code into english.

# Self Documenting Code

Remember SRP? Good, we're done here.

```
if __name__ == "__main__":
    filename_data = "supernovae.txt"
    filename_plot = "cosmology.png"

    supernovae_data = load_data(filename_data)
    chains = fit_cosmology(supernovae_data)
    plot_cosmology(chains, filename_plot)

if __name__ == "__main__":
    directory = "data"
    simulations = [load_simulation(f) for f in get_files(directory)]
    cor = get_correlation_function(simulations)
```

Additionally, each time you feel like you should add a comment, consider refactoring instead. Note in the examples here the overall flow of the program or script is obvious because all the logic and algorithmic trickery is hidden inside functions.

# Inline Comments vs Docstring

Inline comments:

- SPARSE
- Short

Docstring:

- Well structured
- Verbose
- Literally everything

Inline comments should be used sparsely, to either provide a “why” you’re doing something, or to summarise a code block which for some reason hasn’t been refactored out into its own function. Generally the comment is something that \*should\* be its own method, but due to a large number of dependent variables (that would create a massive signature) or a high degree of state (having to pass back a dozen things) makes it better to leave inline.

Docstring is the opposite. For those used to linux and similar, think of docstring like you would `man`.

# The structure of docstring

Focusing on **reStructuredText** docstring here (PEP287), specifically **Numpydoc** syntax.

1. Opening line - one sentence description of method. This is what becomes a tooltip.
2. A more verbose section explaining what the method does. Can include math or visual aid.
3. Details of input parameters - data type, defaults, valid inputs, etc.
4. The return value of the method
5. Potential code examples
6. Potential references
7. Potential outbound links to similar functions

[https://github.com/numpy/numpy/blob/master/doc/HOWTO\\_DOCUMENT.rst.txt](https://github.com/numpy/numpy/blob/master/doc/HOWTO_DOCUMENT.rst.txt)

So what can docstring not do? So note that there are a bunch of valid documentation formats (epytext, reST, Google, Numpydoc, more). Going to focus on numpydoc as its used by numpy, scipy and matplotlib, so very easy to find examples and look at those libraries for reference.

# Very Basics

```
def add(x, y):
    """ The sum of two numbers.

    Adds the two supplied input arguments and returns their summed value.
    The input arguments must be numeric, no type checking is performed.

    Parameters
    -----
    x : numeric
        The first argument to add.
    y : numeric
        The second argument to add.

    Returns
    -----
    numeric
        The summation of `x` and `y`.
    """
    return x + y
```

# Let's get fancy

```
def get_mean_per_bin(x, y, bins=10):
    """ Returns the mean of `y`, binned in `x`.

    Computes the binned mean statistic for `y` for an arbitrary
    number of bins of `x`.

    Parameters
    -----
    x : array-like[numeric]
        The vector to bin.
    y : array-like[numeric]
        The values to take the mean of.
    bins : int, optional
        The number of bins to use, must be a positive integer.
        Defaults to 10.

    Returns
    -----
    np.ndarray
        The bin edges.
    np.ndarray
        The binned averages.

    See Also
    -----
    scipy.stats.binned_statistic: General binned statistics
```

Notes

-----

This method is totally redundant. What is a mean? From [1]:

```
.. math::
    \mu = \frac{1}{N} \sum_{i=0}^N y_i
```

References

-----

[1] A. Child, "Primary School Maths Class", School, 2017.

Examples

-----

```
>>> x, y = np.arange(101), np.arange(101)
>>> get_mean_per_bin(x, y, bins=2)
(array([ 0., 50., 100.]), array([ 24.5, 74.5]))
```

See, simple!

"""

So what can docstring not do? So note that there are a bunch of valid documentation formats (epytext, reST, Google, Numpydoc, more). Going to focus on numpydoc as its used by numpy, scipy and matplotlib, so very easy to find examples and look at those libraries for reference.

**WHO CAN BE BOTHERED?**

# Numpy, Astropy, Scipy, Matplotlib... everyone

```
scipy.interpolate.interp1d
class interp1d(_Interpolator1D):
    """Interpolate f(x,y) known as x,y = f(x). This class returns a function whose call method uses interpolation to find the value of y corresponding to a given value of x.

    If x and y arrays of values used to approximate some function f: ``y = f(x)``. This class returns a function whose call method uses interpolation to find the value of y corresponding to a given value of x.

    Note that calling `interp1d` with NaNs present in input values results in undefined behaviour.

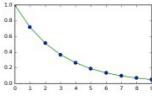
    Parameters
    -----
    x : (N,) array_like
        A 1-D array of real values.
    y : (N,) array_like
        A 1-D array of real values. The length of y along the interpolation axis must be equal to the length of x.
    kind : str or int, optional
        Specifies the kind of interpolation as a string ('linear', 'nearest', 'min', 'spline', 'sod spline', 'takagi' where 'min', 'nearest', 'spline' and 'sod spline' are only supported for 1-D data) or an integer specifying the same.
    axis : int, optional
        Axis along which to interpolate. Interpolation defaults to the last axis of y.
    copy : bool, optional
        If True, make a copy of y (so that it is not modified by this function).
    bounds_error : bool, optional
        If True, raise an error if any points in the interpolation interval lie outside of x and y. Polynomials references to x and y are used. The default is to raise.
    fill_value : array-like or 'extrapolate', optional
        * If a scalar (or float), this value will be used to fill in for requested points outside of the data range. If not provided, then the default is:
            - If 'extrapolate', then points outside the data range will be extrapolated.
            - If 'linear', 'min', 'nearest', 'spline', 'sod spline', 'takagi', then points outside the data range will be filled as if 'fill_value' was provided for x, i.e., ..., fill_value, ..., fill_value.
            - If 'nearest', then points outside the data range will be filled as if 'fill_value' was provided for y, i.e., ..., fill_value, ..., fill_value.
        * If 'linear', 'min', 'nearest', 'spline', 'sod spline', 'takagi', then points outside the data range will be filled as if 'fill_value' was provided for y, i.e., ..., fill_value, ..., fill_value.
    assume_sorted : bool, optional
        If False, values of x can be in any order and they are sorted first. If True, x has to be an array of monotonically increasing values.

    See also
    -----
    interp1d : An object-oriented wrapper of the FITPACK routines.
    interp1d : 2-D interpolator.

    Examples
    -----
    >>> import matplotlib.pyplot as plt
    >>> from scipy import interpolate
    >>> x = np.arange(0, 8)
    >>> y = np.exp(-x)
    >>> f = interpolate.interp1d(x, y)

    >>> new_x = np.arange(0, 8, 0.5)
    >>> new_y = f(new_x) # use interpolation function returned by `interp1d`
    >>> plt.plot(x, y, 'o', new_x, new_y, '-')
    >>> plt.show()

    (Source code)
```



<https://github.com/scipy/scipy/blob/v0.19.1/scipy/interpolate/interpolate.py#L321-L647>

```
class interp1d(_Interpolator1D):
    """
    Interpolate a 1-D function.

    'x' and 'y' are arrays of values used to approximate some function f:
    ``y = f(x)``. This class returns a function whose call method uses
    interpolation to find the value of new points.

    Note that calling `interp1d` with NaNs present in input values results in
    undefined behaviour.

    Parameters
    -----
    x : (N,) array_like
        A 1-D array of real values.
```

Also important to note here that the methods which you expect users to use get the most documentation. Internal methods definitely do not need to have examples and detailed notes!

## **DOCSTRING → DOCS**

### ***TECHNICAL IMPLEMENTATION***

I should probably address how to go from one to the other shouldn't I?

# Sphinx

<http://www.sphinx-doc.org/en/stable/>

<http://www.sphinx-doc.org/en/stable/tutorial.html>

- Write your own doc pages
- Automatically create pages from docstring
- Publish to HTML/PDF/ePub/PlainText

```
pip install Sphinx
```

```
pip install numpydoc
```

```
sphinx-quickstart
```

So two useful links for starting with Sphinx. Installing and setting up a new project is easy, run the quickstart in the top level folder.

# .rst

Sphinx doc pages are .rst - [reStructured Text](#)

See the index.rst, fairly empty.

**make html**

**make latex**

```
.. Test documentation master file, created by
sphinx-quickstart on Mon Nov 13 19:11:02 2017.
You can adapt this file completely to your
liking, but it should at least
contain the root `toctree` directive.
```

```
Welcome to Test's documentation!
=====
```

```
Contents:
```

```
.. toctree::
:maxdepth: 2
```

```
Indices and tables
=====
```

```
* :ref:`genindex`
* :ref:`modindex`
* :ref:`search`
```

So this is the base file that gets generated. Empty page essentially. Can make it as a website and as a pdf for demonstration

.rst

## Table Of Contents

[Welcome to Test's documentation!](#)

[Indices and tables](#)

This Page

[Show Source](#)

Quick search

Go

Enter search terms or a module, class  
or function name.

# Welcome to Test's documentation!

Contents:

## Indices and tables

- [Index](#)
- [Module Index](#)
- [Search Page](#)

And this is the website itself.

# .rst

```
Welcome to Test's documentation!
```

```
Contents:
```

```
.. toctree:::  
    :maxdepth: 2  
  
.. automodule:: misc  
    :members:
```

```
Indices and tables
```

```
* :ref:`genindex`  
* :ref:`modindex`  
* :ref:`search`
```

```
# If extensions (or modules to document with a  
# add these directories to sys.path here. If you  
# documentation root, use os.path.abspath to indicate  
# sys.path.insert(0, os.path.abspath('.'))  
sys.path.insert(0, os.path.abspath('./src'))  
  
# -- General configuration -----  
  
# If your documentation needs a minimal Sphinx  
# needs_sphinx = '1.0'  
  
# Add any Sphinx extension module names here,  
# extensions coming with Sphinx (named 'sphinx_ones').  
extensions = [  
    'sphinx.ext.autodoc',  
    'sphinx.ext.mathjax',  
    'numpydoc'  
]
```

So to get our docstring in there we need to add it to the index.rst, and also tell the newly created conf.py where our code is. Also lets not forget to activate numpydoc in conf.py

.rst

Source code

This Page

Show Source

Quick search

Go

Enter search terms or a module, class  
or function name.

`misc.get_mean_per_bin(x, y, bins=10)`

Returns the mean of  $y$ , binned in  $x$ .

Computes the binned mean statistic for  $y$  for an arbitrary number of bins of  $x$ .

Parameters:

`x : array-like[numeric]`

The vector to bin.

`y : array-like[numeric]`

The values to take the mean of.

`bins : int, optional`

The number of bins to use, must be a positive integer. Defaults to 10.

Returns:

`np.ndarray`

The binned averages.

See also:

`scipy.stats.binned_statistic`

General binned statistics

#### Notes

This method is totally redundant. What even is a mean? From [R1]:

$$\mu = \frac{1}{N} \sum_{i=0}^N y_i$$

This is the output after running make html again

# .rst

```
misc.get_mean_per_bin(x, y, bins=10)
    Returns the mean of y, binned in x.

    Computes the binned mean statistic for y for an arbitrary number of bins of x.

Parameters x : array-like[numeric]
    The vector to bin.

y : array-like[numeric]
    The values to take the mean of.

bins : int, optional
    The number of bins to use, must be a positive integer. Defaults to 10.
```

```
Returns np.ndarray
    The binned averages.
```

**See also:**

`scipy.stats.binned_statistic` General binned statistics

## Notes

This method is totally redundant. What even is a mean? From [RI]:

$$\mu = \frac{1}{N} \sum_{i=0}^N y_i$$

And after compiling the tex from `make latex`

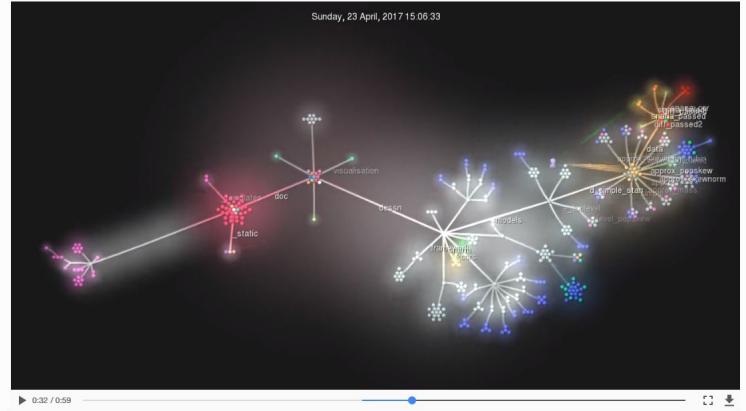
## Extensions - Misc

Restructured Text supports many, many things; Lists, images, tables, links, and also **raw HTML!**

Example from my old doc at <http://deshn.github.io/sn-doc/doc/out/html/index.html>

Visualizing the git commit log

For the project



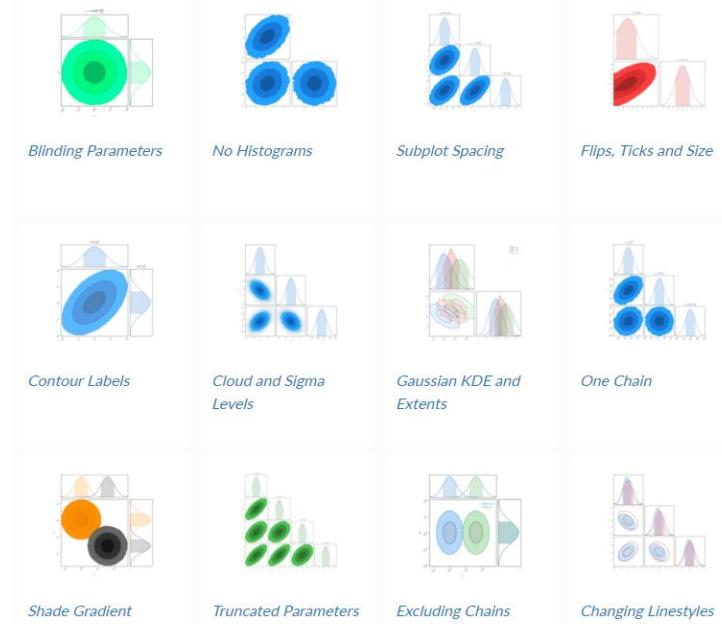
What I want to get across is that you aren't limited by restructured text. You can work directly in html or add in templates (which are html) that other people have created.

# Examples

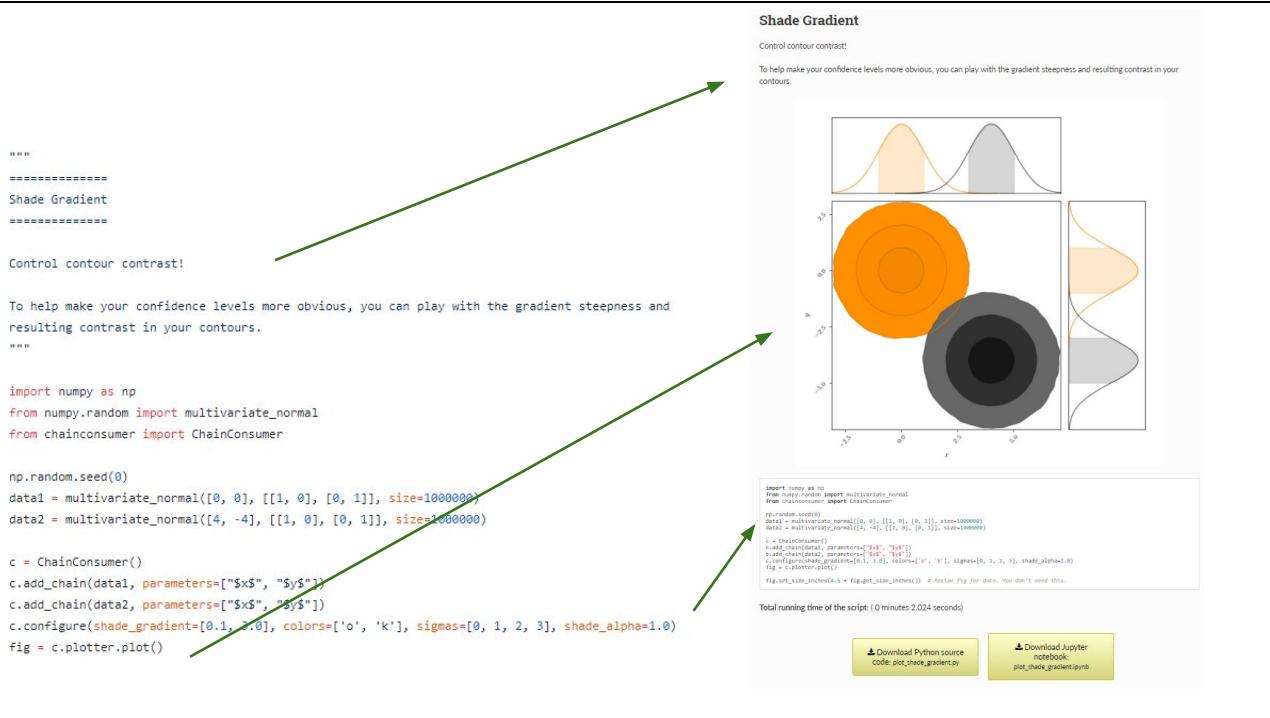
**Sphinx-Gallery**

Write code which gets  
executed and turned into  
examples.

[https://sphinx-gallery.readthe  
docs.io/en/latest/](https://sphinx-gallery.readthedocs.io/en/latest/)



Examples are, in my mind, equally as important as thorough API documentation. There's a reason people go to stackoverflow before reading the official docs - simpler, faster, and normally right to the point with what you need. So sphinx-gallery makes it easy, you can write python files which run as examples, and it will run the code for you, grab the output figures, and throw them into a nice and convenient format.



On the left is the entire file used to create one example. It has some brief docstring at the top (using restructured text's heading format with the === above and below), and the code itself. The figure and download links are created for you.

## And so many more

There are sphinx extensions and themes all over the place. Google them up.

For more basic usage, I recommend comparing the ChainConsumer docs to the source:

<https://samreay.github.io/ChainConsumer/>

<https://github.com/Samreay/ChainConsumer/tree/master/doc>

If you want better examples, dig through scipy and numpy

Fun fact, theres even an extension which will allow you to blog with sphinx. Why someone would do this, I have no idea, but someone coded it up. You can also make html presentations from sphinx, again, why anyone would want this is beyond me, but hey, it exists.

# **Summary**

Documentation is a necessary evil. Please do it.

Sphinx makes it easy, numpydoc has a million examples to make it even simpler.

Will do an interactive sphinx example alongside testing at the end.

However the first part is to add linting. Doesn't really belong here, but balances out the timings.

## BUT FIRST...

### Extending Travis 2

#### *Adding Code Linting*

So its also good to have a code checker and linter. It will alert you to potential bugs and bad practises as you go. The one I use is Codacy.

# Extending Travis - Adding Codacy

Using [Codacy](#)

1

Add project

<https://www.codacy.com>

Samreay / WorkshopExample

GitHub / Public

2

Checking permissions...

As before, other options exist.

3 [Optional] Customise rules

4 [Optional] Add badge

codacy B

Add badge to repository

Badge created

So this part is very easy to setup. You log into Codacy using your github login, add the project, it'll review it and pretty much done. Commit hooks are enabled by default (can go to settings -> integrations to check). In the general settings you can get the markdown to copy and paste to add a badge.

Very simple.

# Extending Travis - Adding Codacy

The screenshot shows the Codacy dashboard for a GitHub repository named 'WorkshopExample'. The repository has one contributor, Samreay, who updated the Codacy file. The code statistics show 9 lines (7 sloc) and 85 Bytes. The .codacy.yml file contains the following content:

```
1 ---  
2 exclude_paths:  
3   - tests/**  
4   - examples/**  
5   - doc/**  
6   - build/**  
7   - dist/**  
8
```

The Project Certification section displays the following scores:

Category	Score
Code Complexity	No Patterns
Compatibility	100%
Error Prone	100%
Security	100%
Code Style	100%
Documentation	No Patterns
Performance	No Patterns
Unused Code	100%

The Issues Breakdown section shows a sun icon and the message "Yay! No Issues".

And this is what the dashboard looks like. Note we're using CodeCov for code coverage, so ignore that panel. As we have such little code, we score well on style, no unused code, no ways to error because theres literally no logic. The security issue outlined is actually an issue in a test file. But wait, we dont want those files! So lets quickly add a .codacy.yml file

Ignore all tests examples, doc, build and dist folder contents, just like we did for the tests. Lots of duplicated folders here, but nothing to do about that.

A Grade code now, what an honour.

# Extending Travis 3

## *Adding Sphinx Documentation*

Also, so lets pause here and have everyone go to <https://www.codacy.com> , log in via github, enable code review on their project. Itll spin on “Reviewing” forever, just refresh and go the main page, make your way to the projects dashboard. And then in project settings you can grab the codacy badge and update the README.md in your project.

# Basic Setup

1. Run sphinx-quickstart like we did before!
2. Add numpydoc and any themes
  - a. I've added the RTD theme and setup a custom CSS override for your convenience.
3. Lets add a new page with the 'API' and set up `automodule` just like we did in the previous Code Documentation section.

The screenshot shows a documentation page for a module named 'some\_module' at version 0.0.1. The top navigation bar is green with the title 'some\_module' and version '0.0.1'. Below it is a search bar. The main content area is black. In the bottom right corner of the main area, there is some small, illegible text. At the very bottom of the page, there is a footer with several links: 'Docs' (with a back arrow), 'Welcome to some\_module's documentation!', 'View page source', 'Contents', 'Indices and tables', and a link to the Sphinx license.

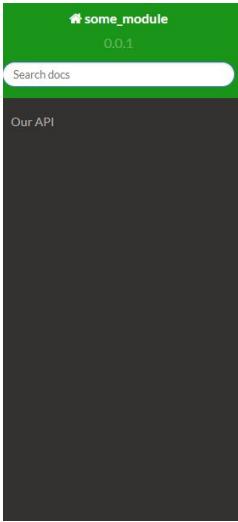
<https://github.com/Samreay/WorkshopExample/commit/e92353a913605713f942c8dc20938989b3ada49f>

This update adds a bit too much code to display. So we run `sphinx-quickstart` as before, change to `numpydoc` as before. You can set the `html` theme in the `conf.py`, which I change from the default to the `read-the-docs` theme, and I add to the `app` setup an override directory, which I place a `css` file in to turn the colour to green you see on the right. (So any future changes can just be done without anyone worrying about configuring `sphinx`).

I also made a new `.rst` file called `api.rst`, and added some comments and the `automodule` declaration just like in the Code Documentation slides. Finally, I also added some comments to the `__init__.py`, including defining `__all__`, and you can see the effect this has on the next slide. Oh, to get `api.rst` to actually get picked up, I added it to the contexts in `index.rst`.

The commit for all these changes is linked at the bottom so you can explicitly go through it.

# Basic Setup



The screenshot shows a documentation page for a project named 'some\_module' version 0.0.1. The top navigation bar includes a search bar labeled 'Search docs' and a link to 'Our API'. The main content area features a heading 'Welcome to some\_module's documentation!' followed by a paragraph of explanatory text. Below this, there are two sections: 'Contents' and 'Indices and tables', each with a list of links. A 'Next' button is visible at the bottom right.

some\_module  
0.0.1

Docs » Welcome to some\_module's documentation!

View page source

## Welcome to some\_module's documentation!

I made this little project so people could easily fork, update the name and author, enable a few web hooks, and have a project which had well defined testing, documentation, code linting and more.

Now I have added a file: `api.rst`, to put the API in (ie the docstring from our code). I include it in the Sphinx build by simply adding it to the contents directly below.

### Contents

- Our API

### Indices and tables

- Index
- Module Index
- Search Page

Next →

This is the new index page

# Basics

Note, without the `_all_` in the `__init__.py`, `add_stuff` is not a module function, it's a function of the file instead.

some\_module  
0.0.1

Search docs

Our API

I made this little project so people could easily fork, update the name and author, enable a few web hooks, and have a project which had well defined testing, documentation, code linting and more.

Here is the docstring from the `__init__.py` file. In this file, it is good practise to define `_all_` so that anyone running `import *` gets the right content!

It is also good for removing unnecessary filenames. If you read the `__init__.py` file, you can see that someone could now run `from some_module import add_stuff` instead of having to run `from some_module.file1 import add_stuff`.

Also note that members without documentation (such as `untested_subtract`) do not appear in the doco).

`some_module.add_stuff(a, b) [source]`

Add something

Parameters:

- `a`: numeric  
First number to add
- `b`: numeric  
Second number to add

Returns:

- numeric  
the sum of the two inputs

Previous

This is the new index page

## Linking to Read-the-docs

1. <https://readthedocs.org>
2. Sign in
3. Link Github / manual project link
4. Add a `.readthedocs.yml`
5. Bask in glory

6 lines (6 sloc) | 121 Bytes

```
1 python:  
2     version: 3.6  
3     setup_py_install: true  
4     extra_requirements:  
5         - numpydoc  
6         - sphinx_rtd_theme
```

<http://workshopexample.readthedocs.io/en/latest/>

So this is how to link to read-the-docs. It'll then monitor your commits and update stuff for you, how nice.

And then at this point, realise that read-the-docs uses project name, so everyone can't just fork this project and have it work. Sigh! Guess we'll do it the dangerous way!!

## **Hosting the HTML output ourselves**

**DANGER DANGER DANGER DANGER DANGER DANGER**

This involves quiet force pushing to our repo on the gh-pages branch.  
If you screw around with the script I give you, you may irreversibly  
wipe your repo. This is bad.

# Hosting the HTML output ourselves

- Travis needs permission to push. Go to your **user** Github, go to settings, Developer settings, Personal Access Tokens and generate a new token.

The screenshot shows the GitHub Developer settings page under the 'Personal access tokens' tab. The left sidebar has three options: 'OAuth Apps', 'GitHub Apps', and 'Personal access tokens', with 'Personal access tokens' being the active tab. The main area is titled 'Personal access tokens' and contains a sub-instruction: 'Tokens you have generated that can be used to access the GitHub API.' Below this, there is a table with one row. The row contains the token name 'Midway Github (Generic repo token) — repo', a timestamp indicating it was last used 'within the last 3 weeks', and two buttons: 'Edit' and 'Delete'. At the top right of the main area, there are two buttons: 'Generate new token' and 'Revoke all'.

Midway Github (Generic repo token) — repo	Last used within the last 3 weeks	Edit	Delete
---	-----------------------------------	------	--------

# Hosting the HTML output ourselves

- 2** Copy the token. DO NOT CLOSE THAT PAGE
- 3** Go to your project in Travis (for me <https://travis-ci.org/Samreay/WorkshopExample>). Go to Settings.
- 4** Add new variable, GITHUB\_API\_KEY, and the value is the token.

## Environment Variables

Notice that the values are not escaped when your builds are executed. Special characters (for bash) should be escaped accordingly.



Anyone that has this key essentially has your account. So you put it as an environment variable so that you don't commit it anywhere, and make sure you do not display its value in the build log.

```
language: python

python:
  - "2.7"
  - "3.6"

env:
  global:
    - GH_REF: github.com/samreay/WorkshopExample.git

install:
  - pip install .
  - pip install -r requirements.txt
```

- 5 Update travis to run a deploy script on success.

And add a GH\_REF variable.

Here is the travis.yml:

```
script:
  - pytest --cov=./ -v

after_success:
  - codecov
  - chmod u+x deploy.sh
  - ./deploy.sh
```

So two changes - the GH\_REF, a global environment variable, and after\_success we run the deploy script

**5** Update travis to run a deploy script on success.  
And add a GH\_REF variable.

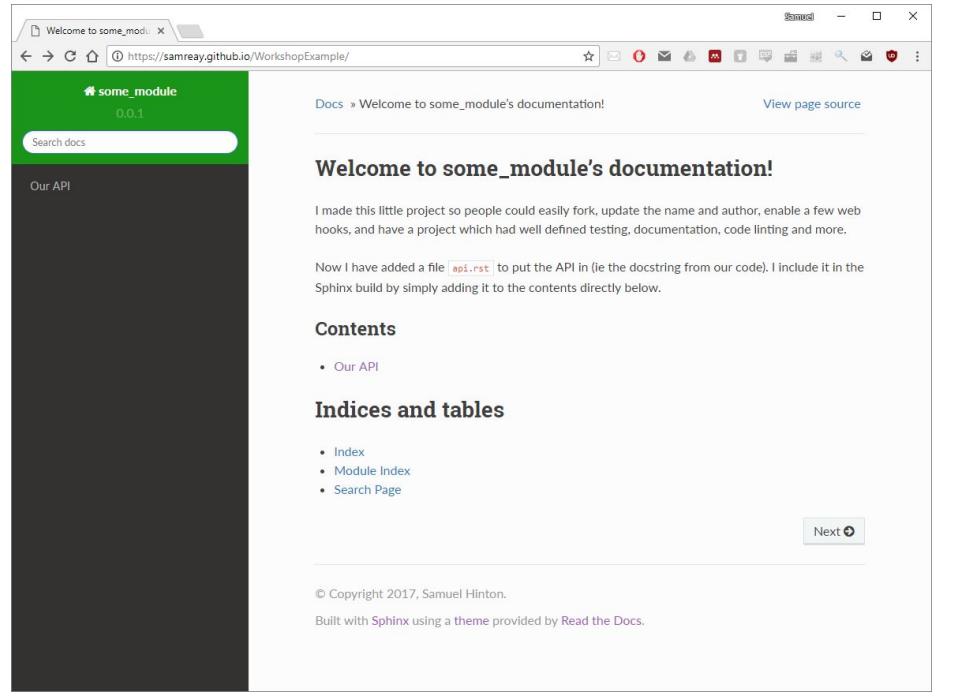
Here is the script:

```
1  #!/usr/bin/env bash
2
3  if [ "$TRAVIS_PULL_REQUEST" != "false" -o "$TRAVIS_BRANCH" != "master" ]; then
4      echo "Not on master branch, nor pull request. Not building doco"
5      exit 0;
6  fi
7
8  if [ -n "$GITHUB_API_KEY" ]; then
9      echo "Github key found. Building documentation."
10     cd "$TRAVIS_BUILD_DIR"/doc
11     make clean
12     make html
13     if [ "$TRAVIS_PYTHON_VERSION" != "3.6" ]; then
14         cd "$TRAVIS_BUILD_DIR"
15         rm -rf .git/
16         cd doc/_build/html
17         git config --global user.email "travis"
18         git config --global user.name "travis"
19         # Create the nojekyll file so Github doesnt try to build for us
20         touch .nojekyll
21         git init
22         git add .
23         echo "Committing Github Pages"
24         git commit -m init
25         # Make sure to make the output quiet, or else the API token will leak!
26         # This works because the API key can replace your password.
27         echo "Pushing"
28         git push -f -q "https://:${GITHUB_API_KEY}@${GH_REF}" master:gh-pages > /dev/null 2>&1 && echo "Pushed"
29     fi
30 fi
31 echo "Deploy script ending"
```

So lets run through the script real quick. Only build the doco if you're on the master branch, or pulling into it. If the key exists that lets us push we go ahead. Will run a make clean in case any files in doc got accidentally committed, and then make html. We will try and do this for all versions to make sure there are no big issues, and if the python version is 3.6 (the latest one) we will push the changes.

If it takes too long, can move the make clean and make html into the python if. We nuke the current git repo, move to the output directory, initialise a new directory, commit everything, and force push it (quiet) to the gh-pages branch, hiding all output so that there's no chance our PAT will end up in the logs.

**Tada!**



**IT LIVES!**

Right, so lets pause here and have everyone set up sphinx interactively. Most of the steps defined before have been done, but what they will need to do is to get that personal access token, set it as an env variable in Travis, and kick off a new build. Maybe useful to point out that if anyone wants to change the styles, I have added a theme\_override.css file in the doc/\_static location for them, can play around with the colour of the top left block as an example.

**End.**

# **Shipping Code**

Get the message out to the masses!

## **The story so far...**

Have a project. Tested.

Documented.

Finally happy for someone else  
to see it!

But... how?

Right, so hopefully this section should be quick so we can do the big interactive thing.  
Will cover PyPI, Zenodo and project distribution methods.

## Go both methods!

### PyPI

Get your packages pip  
installable.

### Zenodo

Get your code archived  
and citable.

It's important to realise PyPI and Zenodo are completely different, so I'll just state it outright. And then move onto PyPI

# PyPI

Register for user: [https://pypi.python.org/pypi?%3Aaction=register\\_form](https://pypi.python.org/pypi?%3Aaction=register_form)

Variety of tutorials:

- <https://blog.jetbrains.com/pycharm/2017/05/how-to-publish-your-package-on-pypi/>
- <https://www.codementor.io/arpitbhayani/host-your-python-package-using-github-on-pypi-du107t7ku>
- <http://peterdowns.com/posts/first-time-with-pypi.html>

I wont do the PyPI stuff interactively, because (unless you're on the test servers), you can't screw around with PyPI. There is no undo, no delete forever, it's serious stuff, so for posterity here are a bunch of useful tutorials before I go over it very quickly myself.

## PyPI

1. Become a PyPI user
2. Setup your `.pypirc` as per the tutorials
3. Ensure your `setup.py` is good (package name, dependencies, it all comes from that).
4. Check you can install it manually - `python setup.py install`
5. Register your package - `python setup.py register`
  - a. `python setup.py register -r https://testpypi.python.org/pypi` (for the test servers)
6. Build your source distribution - `python setup.py sdist`
  - a. (This will be in the dist folder now)
7. Upload your package - `twine upload dist/packagename.zip`
  - a. `twine upload -r test dist/packagename.zip` (if you registered to the test servers)

So there are a few things to note here above the actual steps. Most tutorials will recommend you upload via `python setup.py sdist upload`, but note that this sends your password and username plaintext. Imma pass on that, and twine was created to more securely upload your packages. The other thing to note is that the `setup.py` we have at the moment is very, very minimalistic. There are a bunch of extra fields we can add if we want more detail. The other big thing we should do is manage the versions better, at the moment the version is hardcoded into the `setup.py`, but better if the version is defined in the package (ie `numpy.__version__`) and `setup.py` uses this. You can see how to do that if you look at the `setup.py` for `chainconsumer` - it searches the main file for `__version__` with regex and extracts the version.

# Zenodo

- Versioned *digital* archive
  - Code
  - Datasets
- Generates DOI - makes code **citable**
- Very robust servers
- Integrates with Github
  - <https://guides.github.com/activities/citable-code/>



# Zenodo

1. Log into zenodo, via Github
2. Pick the repository
3. In Github, create a release (which uses git tags)

 Samreay / WorkshopExample

 Code

 Issues 0

 Pull requests 0

 Projects 0

 Wiki

 In

A small github repo used as an example for the coding workshop

[Add topics](#)

 34 commits

 2 branches

 0 releases



There aren't any releases here

Releases are powered by [tagging specific points of history](#) in a repository. They're great for marking release points like `v1.0`.

[Create a new release](#)

# Zenodo

Example:

## Samreay/ChainConsumer: v0.17.0

Samuel Hinton

See readme for changelog.

Preview

**ChainConsumer-v0.17.0.zip**

Name	Size
Samreay-ChainConsumer-cb2a2c8	212 Bytes
└ coveragerc	1.1 kB
└ gitignore	1.5 kB
└ travis.yml	1.1 kB
└ LICENSE	5.7 kB
└ README.md	
└ chainconsumer	110 Bytes
└ __init__.py	16.8 kB
└ analysis.py	31.4 kB
└ chain.py	9.9 kB
└ comparisons.py	5.8 kB
└ diagnostic.py	1.3 kB
└ helpers.py	38.7 kB
└ plotter.py	1.0 kB
└ deploy.sh	6.8 kB
└ doc	
└ Makefile	
└ static	814 Bytes
└ theme_override.css	

Files (893.3 kB)

Name	Size
Samreay/ChainConsumer-v0.17.0.zip md5:bdbef568554abfc9cb3771893801fb81	893.3 kB

[Preview](#) [Download](#)



Publication date:  
June 2, 2017

DOI:  
[DOI 10.5281/zenodo.802290](https://doi.org/10.5281/zenodo.802290)

Related identifiers:  
[Supplement to: https://github.com/Samreay/ChainConsumer/tree/v0.17.0](https://github.com/Samreay/ChainConsumer/tree/v0.17.0)

License (for files):  
 Other (Open)

Versions

Version	Date
v0.17.0 10.5281/zenodo.802290	Jun 2, 2017
v0.16.0 10.5281/zenodo.312352	Feb 21, 2017
v0.15.4 10.5281/zenodo.208329	Dec 19, 2016
v0.14.0 10.5281/zenodo.198252	Dec 9, 2016

# **Project Distribution**

## Project services

1. Set up an environment
2. Install dependencies
- 3. Install project**
4. Download data
5. Run on the data
6. Create output products

We've focused on getting step 3 down pat. But what about the other steps?

# Project services

## Anaconda Projects

*Anaconda Project encapsulates data science projects and makes them easily portable. Project automates setup steps such as installing the right packages, downloading files, setting environment variables and running commands.*

## Docker

*The Docker platform is the only container platform to build, secure and manage the widest array of applications from development to production both on premises and in the cloud*

# Project services

## Anaconda Projects

- Python focused
- Relatively new
- Comes with conda install
- Integrated Jupyter notebook support

## Docker

- Generic applications, not just python
- Highly used and well tested
- Separate install
- Jupyter notebook support with right image

# Project services

## Anaconda Projects Example

<https://blog.daftcode.pl/how-to-wrap-your-repo-with-anaconda-project-c7ee2259ec42>

## Docker Example

<https://docs.docker.com/samples/library/python/#create-a-dockerfile-in-your-python-app-project>

For setting up iPython notebook:

<https://www.dataquest.io/blog/docker-data-science/>

# **Enough of that**

Because we simply dont have the time or bandwidth to give an interactive example of a full pipeline!

## **Where we at**

Have set up a python framework project which has a build pipeline including:

1. Code testing
2. Code coverage
3. Code style
4. Code documentation
5. (And now you know how to PyPI/Zenodo it)

A good example here is to have people make a commit, tag it and push the tag (not in pycharm when you go to push you need to hit the checkbox to push tags as well). It should show up in the Github Releases page on their project (just as a tag) and you can turn that into a release with the draft release. After the link zenodo, that button is the one that will archive it.

If there is time left, it would be beneficial to get people to apply the coding practises/tests/doco principles learnt onto the deliberately bad code I've thrown in one of the .py files in the example project. Fix the code, write tests and doco and get everyone their own project/website running. If anyone has their own code instead of the example code, even better to use that.

After that, we are done! Huzzah!